Searching for Jumbled Patterns in Strings

Ferdinando Cicalese[1], Gabriele Fici[1], and Zsuzsanna Lipták[2]

[1] Dipartimento di Informatica ed Applicazioni, University of Salerno, Italy
{cicalese,fici}@dia.unisa.it
[2] AG Genominformatik, Technische Fakultät, Bielefeld University, Germany
zsuzsa@cebitec.uni-bielefeld.de

Abstract. The Parikh vector of a string s over a finite ordered alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ is defined as the vector of multiplicities of the characters, i.e. $p(s) = (p_1, \ldots, p_\sigma)$, where $p_i = |\{j \mid s_j = a_i\}|$. Parikh vector q occurs in s if s has a substring t with $p(t) = q$. The problem of searching for a query q in a text s of length n can be solved simply and optimally with a sliding window approach in $O(n)$ time. We present two new algorithms for the case where the text is fixed and many queries arrive over time. The first algorithm finds all occurrences of a given Parikh vector in a text (over a fixed alphabet of size $\sigma \geq 2$) and appears to have a sub-linear expected time complexity. The second algorithm only decides whether a given Parikh vector appears in a binary text; it iteratively constructs a linear size data structure which then allows answering queries in constant time, for many queries even during the construction phase.

Keywords: Parikh vectors, permuted strings, pattern matching, string algorithms, average case analysis

1 Introduction

Parikh vectors of strings count the multiplicity of the characters. They have been reintroduced many times by many different names (compomer [5], composition [3], Parikh vector [14], permuted string [7], permuted pattern [9], and others). They are very natural objects to study, if for nothing else because of the many different applications they appear in; for instance, in computational biology, they have been applied for alignment [3], SNP discovery [5], repeated pattern discovery [9], and, most naturally, in interpretation of mass spectrometry data [4]. Parikh vectors can be seen as a generalization of strings, where we view two strings as equivalent if one can be turned into the other by permuting its characters; in other words, if the two strings have the same Parikh vector.

The problem we are interested in here is answering the question whether a query Parikh vector q appears in a given text s (decision version), or where it occurs (occurrence version). An occurrence of q is defined as an occurrence of a substring t of s with Parikh vector q. The problem can be viewed as an approximate pattern matching problem: We are looking for an occurrence of a jumbled version of a query string t, i.e. for the occurrence of a substring t' which has the same Parikh vector. In the following, let n be the length of the text s, m the length of the query q (defined as the length of a string t with Parikh vector q), and σ the size of the alphabet.

The above problem (both decision and occurrence versions) can be solved with a simple sliding window based algorithm, in $O(n)$ time and $O(\sigma)$ additional storage space. This is worst case optimal with respect to the case of one query. However, when we expect to search for many queries in the same string, the above approach leads to $O(Kn)$ runtime for K queries. To the best of our knowledge, no faster approach is known. This is in stark contrast to the classical exact pattern matching problem:

Ferdinando Cicalese, Gabriele Fici, Zsuzsanna Lipták: Searching for Jumbled Patterns in Strings, pp. 105–117.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

There, for one query, any naive approach leads to $O(nm)$ runtime, while quite involved ideas for preprocessing and searching are necessary to achieve an improved runtime of $O(n+m)$, as do the Knuth-Morris-Pratt [12], Boyer-Moore [6] and Boyer-Moore-type algorithms (see, e.g., [2,10]). However, when many queries are expected, the text can be preprocessed to produce a data structure of size linear in n, such as a suffix tree, suffix array, or suffix automaton, which then allows to answer individual queries in time linear in the length of the pattern.

In this paper, we present two new algorithms which perform significantly better than the naive window algorithm, in the case where many queries arrive. In the course of both algorithms, a data structure of size $O(n)$ is constructed, which is subsequently used for fast searching.

1. For general alphabets: We present the Jumping algorithm (Sect. 3) which uses $O(n)$ space to answer occurrence queries in time $O\left(\sigma J \log_2\left(\frac{n}{J}+m\right)\right)$, where J denotes the number of iterations of the main loop of the algorithm. We argue that the expected value of J for the case of random strings and patterns is $O(n/\sqrt{\sigma m})$, yielding an expected runtime of $O\left(\frac{\sqrt{\sigma}\log_2 m}{\sqrt{m}}n\right)$. Our simulations on random strings and real biological strings indicate that this is indeed the performance of the algorithm in practice. This is a significant improvement over the naive algorithm w.r.t. expected runtime, both for a single query and repeated queries over one string.
2. For binary alphabets: After a data structure of size $O(n)$ has been constructed, we answer decision queries in $O(1)$ time (Interval Algorithm, Sect. 4).

The Jumping algorithm is reminiscent of the Boyer-Moore-like approaches to the classical string matching problem [6,2,10]. This analogy is used both in its presentation and in the analysis of the number of iterations performed by the algorithm. We approximate the behavior of the algorithm with a probabilistic automaton, as it is done in [15] to estimate the expected running time of Boyer-Moore on random strings.

A straightforward implementation of the Interval Algorithm requires $\Theta(n^2)$ time for the preprocessing. Instead we present it employing lazy computation of the data structure, and thus the runtime is improved such that a query can be answered either in $O(1)$ or $\Theta(n)$ time, depending on whether the respective entries in the data structure have already been computed. For $K = \omega(n)$ queries, we require $\Theta(K+n^2)$ time (with either implementation), thus always outperforming the naive algorithm, which has $\Theta(Kn)$ runtime. We conjecture that there is no algorithm that can answer any $\Omega(n)$ queries in $o(n^2)$ time.

Related work: An efficient algorithm for computing all Parikh fingerprints of substrings of a given string was developed in [1]. Parikh fingerprints are Boolean vectors where the k'th entry is 1 if and only if a_k appears in the string. The algorithm involves storing a data point for each Parikh fingerprint, of which there are at most $O(n\sigma)$ many. This approach was adapted in [9] for Parikh vectors and applied to identifying all repeated Parikh vectors within a given length range; using it to search for queries of arbitrary length would imply using $\Omega(P(s))$ space, where $P(s)$ denotes the number of different Parikh vectors of substrings of s. This is not desirable, since there are strings with quadratic $P(s)$ [8].

The authors of [7] present an algorithm for finding all occurrences of a Parikh vector in a runlength encoded text. The algorithm's time complexity is $O(n' + \sigma)$, where n' is the length of the runlength encoding of s. Obviously, if the string is not runlength encoded, a preprocessing phase of time $O(n)$ has to be added. However, this may still be feasible if many queries are expected. To the best of our knowledge, this is the only algorithm that has been presented for the problem we are treating here.

2 Notation and Problem Statement

Let $\Sigma = \{a_1, \ldots, a_\sigma\}$ be a finite ordered alphabet. For a string $s \in \Sigma^*$, $s = s_1 \cdots s_n$, we define the Parikh vector $p(s) = (p_1, \ldots, p_\sigma)$ by $p_i := |\{j \mid s_j = a_i\}|$, for $i = 1, \ldots, \sigma$. A Parikh vector p *occurs* in string s if there are positions $i \leq j$ such that $p(s_i \cdots s_j) = p$. We refer to the pair (i, j) as an occurrence of p in s. By convention, we say that the empty string ϵ occurs in each string once. For a Parikh vector $p \in \mathbb{N}^\sigma$, where $\mathbb{N}$ denotes the set of non-negative integers, let $|p| := \sum_i p_i$ denote the *length* of p, namely the length of any string t with $p(t) = p$. Further, by $s[i, j] = s_i \cdots s_j$ we denote the substring of s from i to j, for $1 \leq i \leq j \leq n$.

For two Parikh vectors $p, q \in \mathbb{N}^\sigma$, we define $p \leq q$ and $p + q$ component-wise: $p \leq q$ if and only if $p_i \leq q_i$ for all $i = 1, \ldots, \sigma$, and $p + q = u$ where $u_i = p_i + q_i$ for $i = 1, \ldots, \sigma$. Similarly, for $p \leq q$, we set $q - p = v$ where $v_i = q_i - p_i$ for $i = 1, \ldots, \sigma$.

We want to solve the following problem:

Problem Statement: Let $s \in \Sigma^*$ be given. For a Parikh vector $q \in \mathbb{N}^\sigma$,

1. Decide whether q occurs in s (decision problem);
2. Find all occurrences of q in s (occurrence problem).

In the following, let $|s| = n$ and $|q| = m$. Assume that K many queries arrive over time.

For $K = 1$, both the decision version and the occurrence version can be solved optimally with the following simple algorithm: Move a sliding window of size $|q|$ along string s. This way, we encounter all substrings, and thus all Parikh vectors, of length $|q|$. We maintain the Parikh vector c of the current substring and a counter r which equals the number of indices i such that $c_i \neq q_i$. Each sliding step now costs either 0 or 2 update operations of c, depending on whether the new character entering the window is the same or different from the one that falls out. Whenever we change the value of an entry c_i, we check whether $c_i = q_i$ and increment or decrement r accordingly.

This algorithm solves both the decision and occurrence problems and has running time $\Theta(n)$, using additional storage space $\Theta(\sigma)$. In other words, for one query, it is optimal (save maybe for the additional storage of $\Theta(\sigma)$).

Obviously, one can precompute all sub-Parikh vectors of s, store them (sorted, e.g. lexicographically) and do binary search when a query arrives. Preprocessing time is $\Theta(n^2 \log n)$, because the number of Parikh vectors of s is at most $\binom{n}{2} = O(n^2)$, and there are nontrivial strings with quadratic number of Parikh vectors over arbitrary alphabets [8]. (Now and in the following, we denote the binary logarithm by log, the natural logarithm by ln, and otherwise explicitly state the base.) Moreover, simulations reported there have shown that protein strings have quadratically many

sub-Parikh vectors, a result relevant for mass spectrometry applications. Query time is $O(\log n)$ for the decision problem and $O(\log n + M)$ for the occurrence problem for a query with M occurrences. However, the storage space of $\Theta(n^2)$ is inacceptable in many applications.

For small queries, the problem can be solved exhaustively with a linear size indexing structure such as a suffix tree (size $O(n)$). We can search up to length $m = |q|$ (of the substrings); whenever we find a match, we traverse the subtree below and report the leaf numbers, yielding the occurrences of that substring. Total running time is $O(\sigma^m)$ for searching the tree down to level m, and $O(M)$ total time for the enumeration of the leaves in the individual subtrees, where M is the number of occurrences of q in s. If m is small, namely $m = o(\log_\sigma n)$, then the query time is $o(n) + O(M)$. The suffix tree can be constructed in a preprocessing step in time $O(n)$, so altogether we get time $O(n)$, since $M = O(n)$ for any query q.

3 The Jumping Algorithm

In this section, we introduce our algorithm for general alphabets. Let $s = s_1 \cdots s_n \in \Sigma^*$ be given, and let $pr(i)$ denote the Parikh vector of the prefix of s of length i, for $i = 0, \ldots, n$, where $pr(0) = p(\epsilon) = (0, \ldots, 0)$. We make the following observations:

Observation 1. *Consider Parikh vector $p \in \mathbb{N}^\sigma$, $p \neq (0, \ldots, 0)$.*

1. *For any $1 \leq i \leq j \leq n$, $p = pr(j) - pr(i-1)$ if and only if p occurs in s at position (i, j).*
2. *If an occurrence of p ends in position j, then $pr(j) \geq p$.*

The algorithm moves two pointers L and R along the text, pointing at these potential positions $i - 1$ and j. Instead of moving linearly, however, the pointers are updated in jumps, alternating between updates of R and L, in such a manner that many positions are skipped. Moreover, because of the way we update the pointers, after any update it suffices to check whether $R - L = |q|$ to confirm that an occurrence has been found.

We use the following rules for updating the two pointers, illustrated in Fig. 1:

1. the *first fit rule* for updating R, and
2. the *good suffix rule* for updating L.

First fit rule: Assume that the left pointer is pointing at position L, i.e. no unreported occurrence starts before $L + 1$. Notice that, if there is an occurrence of q ending at any position $j > L$, it must hold that $pr(L) + q \leq pr(j)$. In other words, we must fit both $pr(L)$ and q at position j. We define a function FIRSTFIT as the first potential position where an occurence of a Parikh vector p can end:

$$\text{FIRSTFIT}(p) := \min\{j \mid pr(j) \geq p\}, \tag{1}$$

and set $\text{FIRSTFIT}(p) = \infty$ if no such j exists. We will update R to the first position where $pr(L)$ and q can fit:

$$R \leftarrow \text{FIRSTFIT}(pr(L) + q). \tag{2}$$

Good suffix rule: Now assume that R has just been updated. Thus, $p(s[L+1, R]) = pr(R) - pr(L) \geq q$. If equality holds, then we have found an occurrence of q in position $(L + 1, R)$, and L can be incremented by 1. Otherwise $pr(R) - pr(L) > q$, which implies that, interspersed between the characters that belong to q, there are some "superfluous" characters. Now the first position where an occurrence of q can start is at the beginning of a *contiguous* sequence of characters ending in R which all belong to q. In other words, we need the beginning of the longest suffix of $s[L + 1, R]$ with Parikh vector $\leq q$, i.e. the smallest position i such that $pr(R) - pr(i) \leq q$. We find this position by setting

$$L \leftarrow \text{FIRSTFIT}(pr(R) - q). \tag{3}$$

Note that this rule can also be interpreted as a *bad character rule:* $pr(R) - q = pr(L) + (pr(R) - pr(L)) - q$ contains all those superfluous characters between $L + 1$ and R that we have to fit before a possible next occurrence of q. Below we give the pseudo-code of the algorithm.

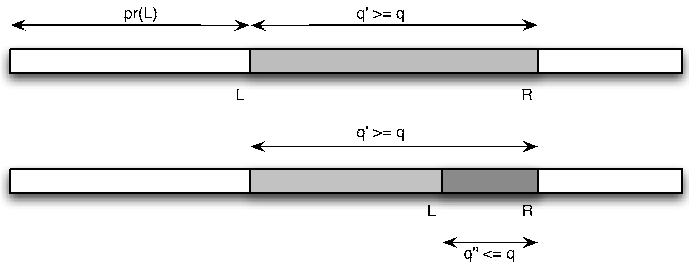

Figure 1. The situation after the update of R (above) and after the update of L (below). R is placed at the first fit of $pr(L) + q$, thus q' is a super-Parikh vector of q. Then L is placed at the beginning of the longest good suffix ending in R, so q'' is a sub-Parikh vector of q.

Algorithm *Jumping Algorithm*
Input: query Parikh vector q
Output: A set Occ containing all beginning positions of occurrences of q in s

```
1.  set m ← |q|; Occ ← ∅; L ← 0;
2.  while L < n − m
3.      do R ← FIRSTFIT(pr(L) + q);
4.         if R − L = m
5.            then add L + 1 to Occ;
6.                 L ← L + 1;
7.            else L ← FIRSTFIT(pr(R) − q);
8.                 if R − L = m
9.                    then add L + 1 to Occ;
10.                        L ← L + 1;
11. return Occ;
```

It remains to see how to compute the FIRSTFIT and pr functions.

3.1 How to compute FIRSTFIT and *pr*

In order to compute FIRSTFIT(p) for some Parikh vector p, we need to know the prefix vectors of s. However, storing all prefix vectors of s would require $O(\sigma n)$ storage space, which may be too much. Instead, the algorithm uses an "inverted prefix vector table" I containing the increment positions of the prefix vectors: for each character $a_k \in \Sigma$, and each value j up to $p(s)_k$, the position in s of the j'th occurrence of character a_k. In other words, $I[k][j] = \min\{i \mid pr(i)_k \geq j\}$ for $j \geq 1$, and $I[k][0] = 0$. Thus we have

$$\text{FIRSTFIT}(p) = \max_{k=1,\ldots,\sigma} \{I[k][p_k]\}. \tag{4}$$

Moreover, we can compute the prefix vectors $pr(i)$ from table I: For $k = 1, \ldots, \sigma$,

$$pr(j)_k = \begin{cases} 0 & \text{if } j < I[k][1] \\ \max\{i \mid I[k][i] \leq j\} & \text{otherwise.} \end{cases} \tag{5}$$

The obvious way to find these values is to do binary search for j in each row of I. However, this would take time $\Theta(\sigma \log n)$; a better way is to use information already acquired during the run of the algorithm. As we shall see later (Lemma 3), it always holds that $L \leq R$. Thus, for computing $pr(R)_k$, it suffices to search for R between $pr(L)_k$ and $pr(L)_k + (R - L)$. This search takes time proportional to $\log(R - L)$. Moreover, after each update of L, we have $L \geq R - m$, so when computing $pr(L)_k$, we can restrict the search for L to between $pr(R)_k - m$ and $pr(R)_k$, in time $O(\log m)$. For more details, see Section 3.4.

Example 2. Let $\Sigma = \{a, b, c\}$ and $s = cabcccaaabccbaacca$. The prefix vectors of s are given below. Note that the algorithm does not actually compute these.

pos.		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
s		*c*	*a*	*b*	*c*	*c*	*c*	*a*	*a*	*a*	*b*	*c*	*c*	*b*	*a*	*a*	*c*	*c*	*a*
# *a*'s	0	0	1	1	1	1	1	2	3	4	4	4	4	4	5	6	6	6	7
# *b*'s	0	0	0	1	1	1	1	1	1	1	2	2	2	3	3	3	3	3	3
# *c*'s	0	1	1	1	2	3	4	4	4	4	4	5	6	6	6	6	7	8	8

The inverted prefix table I:

	0	1	2	3	4	5	6	7	8
a	0	2	7	8	9	14	15	18	
b	0	3	10	13					
c	0	1	4	5	6	11	12	16	17

Query $q = (3, 1, 2)$ has 4 occurrences, beginning in positions $5, 6, 7, 13$, since $(3, 1, 2) = pr(10) - pr(4) = pr(11) - pr(5) = pr(12) - pr(6) = pr(18) - pr(12)$. The values of L and R are given below:

k, see Sec. 3.3	1	2	3	4	5	6	7
L	0	4	5	6	7	10	12
R	8	10	11	12	14	18	18
occ. found?	–	yes	yes	yes	–	–	yes

3.2 Preprocessing

Table I can be computed in one pass over s (where we take the liberty of identifying character $a_k \in \Sigma$ with its index k). The variables c_k count the number of occurrences of character a_k seen so far, and are initialized to 0.

Algorithm *Preprocess* s
1. **for** $i = 1$ **to** n
2. $\quad c_{s_i} = c_{s_i} + 1;$
3. $\quad I[s_i][c_{s_i}] = i;$

Table I requires $O(n)$ storage space (with constant 1). Moreover, the string s can be discarded, so we have zero additional storage.

3.3 Correctness

We have to show that (1) if the algorithm reports an occurrence, then it is correct, and (2) if there is an occurrence, then the algorithm will find it. We first need the following lemma:

Lemma 3. *The following algorithm invariants hold:*

1. *After each update of R, we have $pr(R) - pr(L) \geq q$.*
2. *After each update of L, we have $pr(R) - pr(L) \leq q$.*
3. *$L \leq R$.*

Proof. 1. follows directly from the definition of FIRSTFIT and the update rule for R. For *2.*, if an occurrence was found at (i, j), then before the update we have $L = i - 1$ and $R = j$. Now L is incremented by 1, so $L = i$ and $pr(R) - pr(L) = q - e_{s_i} < q$, where e_k is the k'th unity vector. Otherwise, $L \leftarrow \text{FIRSTFIT}(pr(R) - q)$, and again the claim follows directly from the definition of FIRSTFIT. For *3.*, if an occurrence was found, then L is incremented by 1, and $R - L = m - 1 \geq 0$. Otherwise, $L = \text{FIRSTFIT}(pr(R) - q) = \min\{\ell \mid pr(\ell) \geq pr(R) - q\} \leq R$. □

Proof of (1): If the algorithm reports an index i, then $(i, i+m-1)$ is an occurrence of q: An index i is added to Occ whenever $R - L = m$. If the last update was that of R, then we have $pr(R) - pr(L) \geq q$ by Lemma 3, and together with $R - L = m = |q|$, this implies $pr(R) - pr(L) = q$, thus $(L + 1, R) = (i, i + m - 1)$ is an occurrence of q. If the last update was L, then $pr(R) - pr(L) \leq q$, and it follows analogously that $pr(R) - pr(L) = q$.

Proof of (2): All occurrences of q are reported: Let's assume otherwise. Then there is a minimal i and $j = i + m - 1$ such that $p(s[i, j]) = q$ but i is not reported by the algorithm. By Observation 1, we have $pr(j) - pr(i - 1) = q$.

Let's refer to the values of L and R as two sequences $(L_k)_{k=1,2,\ldots}$ and $(R_k)_{k=1,2,\ldots}$. So we have $L_1 = 0$, and for all $k \geq 1$, $R_k = \text{FIRSTFIT}(pr(L_k) + q)$, and $L_{k+1} = L_k + 1$ if $R_k - L_k = m$ and $L_{k+1} = \text{FIRSTFIT}(pr(R_k) - q)$ otherwise. In particular, $L_{k+1} > L_k$ for all k.

First observe that if for some k, $L_k = i - 1$, then R will be updated to j in the next step, and we are done. This is because $R_k = \text{FIRSTFIT}(pr(L_k) + q) = \text{FIRSTFIT}(pr(i - 1) + q) = \text{FIRSTFIT}(pr(j)) = j$. Similarly, if for some k, $R_k = j$, then we have $L_{k+1} = i - 1$.

So there must be a k such that $L_k < i-1 < L_{k+1}$. Now look at R_k. Since there is an occurrence of q after L_k ending in j, this implies that $R_k = \text{FIRSTFIT}(pr(L_k)+q) \leq j$. However, we cannot have $R_k = j$, so it follows that $R_k < j$. On the other hand, $i-1 < L_{k+1} \leq R_k$ by our assumption and by Lemma 3. So R_k is pointing to a position somewhere between $i-1$ and j, i.e. to a position within our occurrence of q. Denote the remaining part of q to the right of R_k by q': $q' = pr(j) - pr(R_k)$. Since $R_k = \text{FIRSTFIT}(pr(L_k) + q)$, all characters of q must fit between L_k and R_k, so the Parikh vector $p = pr(i) - pr(L_k)$ is a super-Parikh vector of q'. If $p = q'$, then there is an occurrence of q at $(L_k + 1, R_k)$, and by minimality of (i, j), this occurrence was correctly identified by the algorithm. Thus, $L_{k+1} = L_k + 1 \leq i - 1$, contradicting our choice of k. It follows that $p > q'$ and we have to find the longest good suffix of the substring ending in R_k for the next update L_{k+1} of L. But $s[i, R_k]$ is a good suffix because its Parikh vector is a sub-Parikh vector of q, so $L_{k+1} = \text{FIRSTFIT}(pr(R_k) - q) \leq i - 1$, again in contradiction to $L_{k+1} > i - 1$.

We illustrate the proof in Fig. 2.

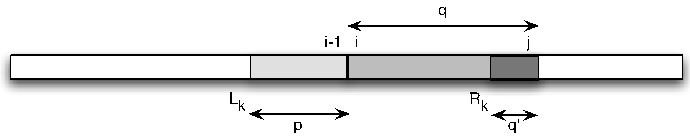

Figure 2. Illustration for proof of correctness.

3.4 Algorithm Analysis

Let $\mathbb{A}(s, q)$ denote the running time of Jumping Algorithm over a text s and a Parikh vector q. Let $J = J(s, q)$ be the number of iterations performed in the `while` loop in line 2, i.e., the number of jumps performed by the algorithm, for the input (s, q). Further, for each $i = 1, \ldots, J$, let $\hat{L}_i$, $\hat{R}_i$ denote the value of L and R, respectively, after the i'th execution of line 3 of the algorithm[1].

In order to calculate the running time of the algorithm on the given input we need to evaluate the number of iterations it performs, the running time of the functions FIRSTFIT and the time needed to compute the Parikh vectors $pr(\cdot)$ necessary in lines 3 and 7.

It is easy to see that computing FIRSTFIT takes $O(\sigma)$ time.

The computation of $pr(\hat{L}_i)$ in line 3 takes $O(\sigma \log m)$: For each $k = 1, \ldots, \sigma$, the component $pr(\hat{L}_i)_k$ can be determined by binary search over the list $I[k][pr(\hat{R}_{i-1})_k - m], I[k][pr(\hat{R}_{i-1})_k - m + 1], \ldots, I[k][pr(\hat{R}_{i-1})_k]$. By $\hat{L}_i \geq \hat{R}_{i-1} - m$, the claim follows.

The computation of $pr(\hat{R}_i)$ in line 7 takes $O(\sigma \log(\hat{R}_i - \hat{R}_{i-1} + m))$. Simply observe that in the prefix ending at position $\hat{R}_i$ there can be at most $\hat{R}_i - \hat{L}_i$ more occurrences of the k'th character than there are in the prefix ending at position $\hat{L}_i$. Therefore, as before, we can determine $pr(\hat{R}_i)_k$ by binary search over the list $I[k][pr(\hat{L}_i)_k], I[k][pr(\hat{L}_i)_k + 1], \ldots, I[k][pr(\hat{L}_i)_k + \hat{R}_i - \hat{L}_i]$. Using the fact that $\hat{L}_i \geq \hat{R}_{i-1} - m$, the desired bound follows.

[1] The $\hat{L}_i$ and $\hat{R}_i$ coincide with the L_k and R_k from Section 3.3 almost but not completely: When an occurrence is found after the update of L, then the corresponding pair L_k, R_k is skipped here. The reason is that now we are only considering those updates that carry a computational cost.

The last three observations imply

$$\mathbb{A}(s,q) = O\left(\sigma J \log m + \sigma \sum_{i=1}^{J} \log(\hat{R}_i - \hat{R}_{i-1} + m)\right).$$

Note that this is an overestimate, since line 7 is only executed if no occurrence was found after the current update of R (line 4). Standard algebraic manipulations using Jensen's inequality (see, e.g. [11]) yield $\sum_{i=1}^{J} \log(\hat{R}_i - \hat{R}_{i-1} + m) \leq J \log\left(\frac{n}{J} + m\right)$. Therefore we obtain

$$\mathbb{A}(s,q) = O\left(\sigma J \log\left(\frac{n}{J} + m\right)\right). \tag{6}$$

The worst case running time of the Jumping Algorithm is superlinear, since there exist strings s and Parikh vectors q such that $J = \Theta(n)$: For instance, on the string $s = ababab\cdots ab$ and $q = (2,0)$, the algorithm will execute $n/2$ jumps.

This sharply contrasts with the experimental evaluation we present later. The Jumping Algorithm appears to have in practice a sublinear behavior. In the rest of this section we sketch an average case analysis of the running time of the Jumping Algorithm leading to the conclusion that its expected running time is sublinear.

We assume that the string s is given as a sequence of i.i.d. random variables uniformly distributed over the alphabet Σ. According to Knuth *et al.* [12] "It might be argued that the average case taken over random strings is of little interest, since a user rarely searches for a random string. However, this model is a reasonable approximation when we consider those pieces of text that do not contain the pattern [...]". The experimental results we provide will show that this is indeed the case.

To simplify the presentation, let us fix the Parikh vector q as being perfectly balanced, i.e., $q = (\frac{m}{\sigma}, \ldots, \frac{m}{\sigma})$. Let E_i denote the expected number ℓ such that $\textsc{firstfit}(pr(i) + q) = i + \ell$. Because of the assumption on the string, we have that E_i is independent of i, so we can write $E_i = E_{m,\sigma}$. In particular, we have

$$E_{m,\sigma} \approx m + \begin{cases} m2^{-m}\binom{m}{m/2} & \text{if } \sigma = 2, \\ \sqrt{2m\sigma \ln \frac{\sigma}{\sqrt{2\pi}}} & \text{otherwise.} \end{cases} \tag{7}$$

This result can be found in [13] where the author studied a variant of the well known coupon collector problem in which the collector has to accumulate a certain number of copies of each coupon. It should not be hard to see that by identifying the characters with the coupon types, the random string with the sequence of coupons obtained, and the query Parikh vector with the number of copies we require for each coupon type, the expected time when the collection is finished is the same as our $E_{m,\sigma}$.

We shall now follow the approach taken by Schaback in the average case analysis of the Boyer-Moore algorithm [15]. We build a probabilistic automaton which simulates the behavior of the Jumping Algorithm. We also assume that each new reference to a position in the string is done by generating the character again. See [15] for how this assumption does not affect the result.

The automaton $\mathcal{A}(n,m,\sigma)$ moves the pointers L and R along the string as follows: with probability $\zeta = \zeta(m,\sigma)$ the pointer L is moved forward by one position (this corresponds to the case of a match); with probability $(1-\zeta)$ the pointer R is moved

forward to the closest position to L such that $pr(R) - pr(L) \geq q$; in this case also L is updated and set to $R - m$ (this corresponds to the case of no match; in fact, we are upper bounding the Jumping Algorithm's behavior, since it always updates L to a position at most m away from R).

Let $\mathbb{E}[\mathcal{A}(n, m, \sigma)]$ denote the expected number of jumps of $\mathcal{A}(n, m, \sigma)$. We have $\mathbb{E}[\mathcal{A}(n, m, \sigma)] = \frac{n}{\zeta + (1-\zeta)(E_{m,\sigma} - m)}$. If we take ζ to be the probability that a random string of size m over an alphabet of size σ has Parikh vector q, we get $\zeta \approx \sqrt{\frac{\sigma^\sigma}{(2\pi m)^{\sigma-1}}}$, where we use Stirling approximation for the multinomial $\binom{m}{\frac{m}{\sigma}, \ldots, \frac{m}{\sigma}}$. Note that due to the magnitude of ζ, for large values of m, we have

$$\mathbb{E}[\mathcal{A}(n, m, \sigma)] \approx n/(E_{m,\sigma} - m). \tag{8}$$

Recalling (6) and using (7) and (8) as an approximation of the number $\mathbb{E}[J]$ of jumps performed by the Jumping Algorithm, over a random instance, we get that the average case complexity of the Jumping Algorithm can be estimated as $O\left(n\sqrt{\frac{2\pi}{m}} \log m\right)$ in the case of a binary alphabet, and $O\left(\frac{\sigma n}{\sqrt{2\sigma m \ln\left(\sigma/\sqrt{2\pi}\right)}} \log m\right)$, for $\sigma \geq 3$. Summarizing, according to the above approximations, we would expect the algorithm's running time to be $O\left(\frac{n \log m}{\sqrt{m}}\right)$, with the constant in the order of $\sqrt{\frac{\sigma}{2 \ln \sigma}}$.

We conclude this section by remarking once more that the above estimate obtained by the approximating probabilistic automaton appears to be confirmed by the experiments.

3.5 Simulations

We implemented the Jumping Algorithm in C++ in order to study the number of jumps J. We ran it on random strings of different lengths and over different alphabet sizes. The underlying probability model is an i.i.d. model with uniform distribution. We sampled random query vectors with length between $\log n$ $(= \log_2 n)$ and $\sqrt{n}$, where n is the length of the string. Our queries were of one of two types:

1. Quasi-balanced Parikh vectors: Of the form $(q_1, \ldots, q_\sigma)$ with $q_i \in (x - \epsilon, x + \epsilon)$, and x running from $\log n/\sigma$ to $\sqrt{n}/\sigma$. For simplicity, we fixed $\epsilon = 10$ in all our experiments, and sampled uniformly at random from all quasi-balanced vectors around each x.
2. Random Parikh vectors with fixed length m. These were sampled uniformly at random from the space of all Parikh vectors with length m.

The rationale for using quasi-balanced queries is that those are clearly worst-case for the number of jumps J, since J depends on the shift length, which in turn depends on FIRSTFIT$(pr(L) + q)$. Since we are searching in a random string with uniform character distribution, we can expect to have minimal FIRSTFIT$(pr(L) + q)$ if q is close to balanced, i.e. if all entries q_i are roughly the same. This is confirmed by our experimental results which show that J decreases dramatically if the queries are not balanced (Fig. 4, right).

We ran experiments on random strings over different alphabet sizes, and observe that our average case analysis agrees well with the simulation results for random strings and random quasi-balanced query vectors. Plots for $n = 10^5$ and $n = 10^6$ with alphabet sizes $\sigma = 2, 4, 16$ resp. $\sigma = 4, 16$ are shown in Fig. 3.

To see how our algorithm behaves on non-random strings, we downloaded human DNA sequences from GenBank (`http://www.ncbi.nlm.nih.gov/Genbank/`) and ran the Jumping Algorithm with random quasi-balanced queries on them. We found that the algorithm performs 2 to 10 times fewer jumps on these DNA strings than on random strings of the same length, with the gain increasing as n increases. We show the results on a DNA sequence of 1 million bp (from Chromosome 11) in comparison with the average over 10 random strings of the same length (Fig. 4, left).

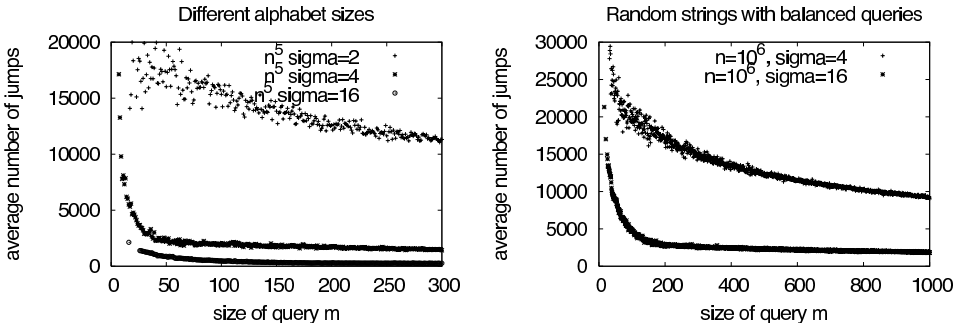

Figure 3. Number of jumps for different alphabet sizes for random strings of size 100 000 (left) and 1 000 000 (right). All queries are randomly generated quasi-balanced Parikh vectors (cf. text). Data averaged over 10 strings and all random queries of same length.

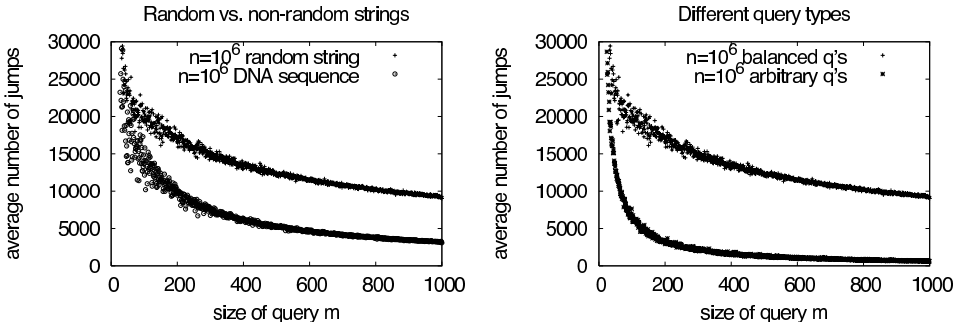

Figure 4. Number of jumps in random vs. nonrandom strings: Random strings over an alphabet of size 4 vs. a DNA sequence, all of length 1 000 0000, random quasi-balanced query vectors. Data averaged over 10 random strings and all queries with the same length (left). Comparison of quasi-balanced vs. arbitrary query vectors over random strings, alphabet size 4, length 1 000 000, 10 strings. The data shown are averaged over all queries with same length m (right).

4 Often Constant Query Time for Binary Alphabets

We now describe our algorithm for binary alphabets. It uses a data structure of size $O(n)$ which it constructs in a lazy manner, only computing those entries that are needed for the current query, and storing them for future queries. Once the data structure has been completely constructed, all queries can be answered in constant time. During the construction phase, answering queries may take either $O(1)$ or $O(n)$ time. Only decision queries are answered.

The algorithm makes use of the following property of binary strings:

Lemma 4. *Let $s \in \{a,b\}^*$ with $|s| = n$. Fix $1 \leq m \leq n$. If the Parikh vectors $(x_1, m-x_1)$ and $(x_2, m-x_2)$ both occur in s, then so does $(y, m-y)$ for any $x_1 \leq y \leq x_2$.*

Proof. Consider a sliding window of fixed size m moving along the string and let (p_1, p_2) be the Parikh vector of the current substring. When the window is shifted by one, the Parikh vector either remains unchanged (if the character falling out is the same as the character coming in), or it becomes (p_1+1, p_2-1) resp. (p_1-1, p_2+1) (if they are different). Thus the Parikh vectors of substrings of s of length m build a set of the form $\{(x, m-x) \mid x = \min(m), \min(m)+1, \ldots, \max(m)\}$ for appropriate $\min(m)$ and $\max(m)$. In other words, they build an interval. □

So all we need in order to decide whether a query $q = (q_1, q_2)$ with $|q| = m$ has an occurrence in s is to check whether $\min(m) \leq q_1 \leq \max(m)$. We would like to have a table with $\min(m)$ and $\max(m)$ for all $1 \leq m \leq n$; however, computing the complete table takes $O(n^2)$ time. Notice though that, for any individual query q, we only need the values for $|q|$. So when a query q arrives with $|q| = m$, we look up whether $\min(m)$ and $\max(m)$ have already been computed. If so, we answer the query in constant time. Otherwise, we compute the entries for m by moving a sliding window of size m over s and collecting the minimum and maximum number of a's.

Analysis: All queries take time either $O(1)$ or $O(n)$, and after n queries of the latter kind, the table is completely constructed and all subsequent queries can be answered in $O(1)$ time. If we assume that the query lengths are uniformly distributed, then we can view this as another coupon collector problem (see Section 3.4), where the coupon collector has to collect one copy of each n coupons, namely the different lengths m. Then the expected number of queries needed before having seen all m and thus before having completed the table is $nH_n \approx n \ln n$. The algorithm will have taken $O(n^2)$ time to answer these $n \ln n$ queries, because it spends linear time only on queries with new length m, and $O(1)$ on queries with length that it has seen before; now it can answer all further queries in constant time.

The assumption of the uniform length distribution may not be very realistic; however, even if it does not hold, we never take more time than $O(n^2 + K)$ for K many queries. Since any one query may take at most $O(n)$ time, our algorithm never performs worse than the naive algorithm. Moreover, for those queries where the table entries have to be computed, we can even run the naive algorithm itself and report all occurrences, as well. For all others, we only give decision answers, but in constant time.

Finally, the table can of course be computed completely in a preprocessing step in $O(n^2)$ time, thus always guaranteeing constant query time. The overall running time is $\Theta(K + n^2)$. As long as the number of queries is $K = \omega(n)$, this variant, too, outperforms the naive algorithm, whose running time is $\Theta(Kn)$.

5 Conclusion and Open Problems

Our simulations appear to confirm that in practice the performance of the Jumping Algorithm is well predicted by the expected $O(\frac{\sqrt{\sigma}\log m}{\sqrt{m}}n)$ time of the probabilistic

analysis we proposed. A more precise analysis is needed, however. Our approach seems unlikely to lead to any refined average case analysis since that would imply improved results for the intricate variant of the coupon collector problem of [13].

Moreover, in order to better simulate DNA or other biological data, more realistic random string models than uniform i.i.d. should also be analysed, such as first or higher order Markov chains.

Another open problem is whether the Interval Algorithm can be improved by constructing in subquadratic time the data structure it uses (in a preprocessing step). In fact, we conjecture that this is not possible, and that no algorithm can answer arbitrary $\Omega(n)$ many queries in $o(n^2)$ time. However, proving such an upper bound has so far proven elusive.

Acknowledgements: Thanks to Travis Gagie for the pointer to some recent literature on the coupon collector problem, and for pointing out the similarities of the Jumping Algorithm with Boyer-Moore. We thank Rosa Caiazzo for generously giving us of her time. Part of this work was done while F. C. and Zs. L. were funded by a Sofja Kovalevskaja grant of the Alexander von Humboldt Foundation and the German Federal Ministry of Education and Research.

References

1. A. Amir, A. Apostolico, G. M. Landau, and G. Satta: *Efficient text fingerprinting via Parikh mapping*. J. Discrete Algorithms, 1(5-6) 2003, pp. 409–421.
2. A. Apostolico and R. Giancarlo: *The Boyer-Moore-Galil string searching strategies revisited*. SIAM J. Comput., 15(1) 1986, pp. 98–105.
3. G. Benson: *Composition alignment*, in Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03), 2003, pp. 447–461.
4. S. Böcker: *Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt*. Journal of Computational Biology, 11(6) 2004, pp. 1110–1134.
5. S. Böcker: *Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry*. Bioinformatics, 23(2) 2007, pp. 5–12.
6. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Commun. ACM, 20(10) 1977, pp. 762–772.
7. A. Butman, R. Eres, and G. M. Landau: *Scaled and permuted string matching*. Inf. Process. Lett., 92(6) 2004, pp. 293–297.
8. M. Cieliebak, T. Erlebach, Zs. Lipták, J. Stoye, and E. Welzl: *Algorithmic complexity of protein identification: combinatorics of weighted strings*. Discrete Applied Mathematics, 137(1) 2004, pp. 27–46.
9. R. Eres, G. M. Landau, and L. Parida: *Permutation pattern discovery in biosequences*. Journal of Computational Biology, 11(6) 2004, pp. 1050–1060.
10. R. N. Horspool: *Practical fast searching in strings*. Softw., Pract. Exper., 10(6) 1980, pp. 501–506.
11. S. Jukna: *Extremal Combinatorics*, Springer, 1998.
12. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
13. R. May: *Coupon collecting with quotas*. Electr. J. Comb., 15 2008.
14. A. Salomaa: *Counting (scattered) subwords*. Bulletin of the European Association for Theoretical Computer Science (EATCS), 81 2003, pp. 165–179.
15. R. Schaback: *On the expected sublinearity of the Boyer-Moore algorithm*. SIAM J. Comput., 17(4) 1988, pp. 648–658.